

03 | 预处理阶段能做什么：宏定义和条件编译

2020-05-12 罗剑锋

罗剑锋的C++实战笔记

[进入课程 >](#)



讲述：Chrono

时长 13:25 大小 12.30M



你好，我是 Chrono。

上一次我讲了在编码阶段要有好的 code style，尽量写出“人能够看懂的代码”。今天，我就继续讲讲编码后的预处理阶段，看看这个阶段我们能做哪些事情。

预处理编程

其实，只要写 C/C++ 程序，就会用到预处理，只是大多数时候，你只用到了它的一点[☆]能。比如，在文件开头写上“#include”这样的语句，或者用“#define”定义一些常^安x。只是这些功能都太简单了，没有真正发挥出预处理器的本领，所以你可能几乎感觉不到它的存在。

预处理只能用很少的几个指令，也没有特别严谨的“语法”，但它仍然是一套完整自治的语言体系，使用预处理也能够实现复杂的编程，解决一些特别的问题——虽然代码可能会显得有些“丑陋”“怪异”。

那么，“预处理编程”到底能干什么呢？

你一定要记住：**预处理阶段编程的操作目标是“源码”，用各种指令控制预处理器，把源码改造成另一种形式，就像是捏橡皮泥一样。**

把上面的这句话多读几遍，仔细揣摩体会一下，理解了之后，你再去用那些预处理指令就会有不一样的感觉了。

C++ 语言有近百个关键字，预处理指令只有十来个，实在是少得可怜。而且，常用的也就是 `#include`、`#define`、`#if`，所以很容易掌握。不过，有几个小点我还是要特别说一下。

首先，预处理指令都以符号“`#`”开头，这个你应该很熟悉了。但同时你也应该意识到，虽然都在一个源文件里，但它不属于 C++ 语言，它走的是预处理器，不受 C++ 语法规则的约束。

所以，预处理编程也就不用太遵守 C++ 代码的风格。一般来说，预处理指令不应该受 C++ 代码缩进层次的影响，不管是在函数、类里，还是在 `if`、`for` 等语句里，永远是**顶格写**。

另外，单独的一个“`#`”也是一个预处理指令，叫“空指令”，可以当作特别的预处理空行。而“`#`”与后面的指令之间也可以有空格，从而实现缩进，方便排版。


下面是一个示例，`#` 号都在行首，而且 `if` 里面的 `define` 有缩进，看起来还是比较清楚的。以后你在写预处理代码的时候，可以参考这个格式。

```
1 #                                // 预处理空行
2 #if __linux__                    // 预处理检查宏是否存在
3 #   define HAS_LINUX            1 // 宏定义，有缩进
4 #endif                          // 预处理条件语句结束
5 #                                // 预处理空行
```

 复制代码

预处理程序也有它的特殊性，暂时没有办法调试，不过可以让 GCC 使用 “-E” 选项，略过后面的编译链接，只输出预处理后的源码，比如：

```
1 g++ test03.cpp -E -o a.cxx      #输出预处理后的源码
```

 复制代码

多使用这种方式，对比一下源码前后的变化，你就可以更好地理解预处理的工作过程了。


这几个小点有些杂，不过你只要记住 “# 开头、顶格写” 就行了。

包含文件 (#include)

先来说说最常用的预处理指令 “#include”，它的作用是 “**包含文件**”。注意，不是 “包含头文件”，而是**可以包含任意的文件**。

也就是说，只要你愿意，使用 “#include” 可以把源码、普通文本，甚至是图片、音频、视频都引进来。当然，出现无法处理的错误就是另外一回事了。

```
1 #include "a.out"      // 完全合法的预处理包含指令，你可以试试
```

 复制代码

可以看到，“#include” 其实是非常 “弱” 的，不做什么检查，就是 “死脑筋” 把数据合并进源文件。

所以，在写头文件的时候，为了防止代码被重复包含，通常要加上 “**Include Guard**”，也就是用 “#ifndef/#define/#endif” 来保护整个头文件，像下面这样：

```
1 #ifndef _XXX_H_INCLUDED_
2 #define _XXX_H_INCLUDED_
3
4 ...    // 头文件内容
5
6 #endif // _XXX_H_INCLUDED_
```

 复制代码

这个手法虽然比较“原始”，但在目前来说（C++11/14），是唯一有效的方法，而且也向下兼容 C 语言。所以，我建议你所有头文件里强制使用。

除了最常用的包含头文件，你还可以利用“#include”的特点玩些“小花样”，编写一些代码片段，存进“*.inc”文件里，然后有选择地加载，用得好的话，可以实现“源码级别的抽象”。

比如说，有一个用于数值计算的大数组，里面有成百上千个数，放在文件里占了很多地方，特别“碍眼”：

复制代码

```
1 static uint32_t calc_table[] = { // 非常大的一个数组，有几十行
2     0x00000000, 0x77073096, 0xee0e612c, 0x990951ba,
3     0x076dc419, 0x706af48f, 0xe963a535, 0x9e6495a3,
4     0x0edb8832, 0x79dcb8a4, 0xe0d5e91e, 0x97d2d988,
5     0x09b64c2b, 0x7eb17cbd, 0xe7b82d07, 0x90bf1d91,
6     ...
7 };
```

这个时候，你就可以把它单独摘出来，另存为一个“*.inc”文件，然后再用“#include”替换原来的大批数字。这样就节省了大量的空间，让代码更加整洁。

复制代码

```
1 static uint32_t calc_table[] = {
2 # include "calc_values.inc" // 非常大的一个数组，细节被隐藏
3 };
```

宏定义（#define/#undef）

接下来要说的是预处理编程里最重要、最核心的指令“#define”，它用来定义一个源码级别的“**文本替换**”，也就是我们常说的“**宏定义**”。

“#define”可谓“无所不能”，在预处理阶段可以无视 C++ 语法限制，替换任何文字，定义常量 / 变量，实现函数功能，为类型起别名（typedef），减少重复代码.....

不过，也正是因为它太灵活，如果过于随意地去使用宏来写程序，就有可能把正常的 C++ 代码搞得“千疮百孔”，替换来替换去，都不知道真正有效的代码是什么样子了。


所以，使用宏的时候一定要谨慎，时刻记着以简化代码、清晰易懂为目标，不要“滥用”，避免导致源码混乱不堪，降低可读性。

下面，我就说几个注意事项，帮你用好宏定义。

首先，因为宏的展开、替换发生在预处理阶段，不涉及函数调用、参数传递、指针寻址，没有任何运行期的效率损失，所以对于一些调用频繁的小代码片段来说，用宏来封装的效果比 inline 关键字要更好，因为它真的是源码级别的无条件内联。


下面有几个示例，摘自 Nginx，你可以作为参考：

```
1 #define ngx_tolower(c)      ((c >= 'A' && c <= 'Z') ? (c | 0x20) : c)
2 #define ngx_toupper(c)     ((c >= 'a' && c <= 'z') ? (c & ~0x20) : c)
3
4 #define ngx_memzero(buf, n) (void) memset(buf, 0, n)
```

 复制代码


其次，你要知道，**宏是没有作用域概念的，永远是全局生效**。所以，对于一些用来简化代码、起临时作用的宏，最好是用完后尽快用“#undef”取消定义，避免冲突的风险。像下面这样：

```
1 #define CUBE(a) (a) * (a) * (a) // 定义一个简单的求立方的宏
2
3 cout << CUBE(10) << endl;      // 使用宏简化代码
4 cout << CUBE(15) << endl;      // 使用宏简化代码
5
6 #undef CUBE                    // 使用完毕后立即取消定义
```

 复制代码

另一种做法是**宏定义前先检查**，如果之前有定义就先 undef，然后再重新定义：

```
1 #ifdef AUTH_PWD                // 检查是否已经有宏定义
2 # undef AUTH_PWD              // 取消宏定义
```

 复制代码

```
3 #endif // 宏定义检查结束
4 #define AUTH_PWD "xxx" // 重新宏定义
```

再次，你可以适当使用宏来定义代码中的常量，消除“魔术数字”“魔术字符串”（magic number）。

虽然不少人认为，定义常量更应该使用 enum 或者 const，但我觉得宏定义毕竟用法简单，也是源码级的真正常量，而且还是从 C 继承下来的传统，用在头文件里还是有些优势的。

这种用法非常普遍，你可能也经常用，我就简单举两个例子吧：

```
1 #define MAX_BUF_LEN 65535
2 #define VERSION "1.0.18"
```

 复制代码

不过你要注意，关键是要“适当”，自己把握好分寸，不要把宏弄得“满天飞”。

除了上面说的三个，如果你开动脑筋，用好“文本替换”的功能，也能发掘出许多新颖的用法。我有一个比较实际的例子，用宏来代替直接定义名字空间（namespace）：

```
1 #define BEGIN_NAMESPACE(x) namespace x {
2 #define END_NAMESPACE(x) }
3
4 BEGIN_NAMESPACE(my_own)
5
6 ... // functions and classes
7
8 END_NAMESPACE(my_own)
```

 复制代码

这里我定义了两个宏：BEGIN_NAMESPACE 和 END_NAMESPACE，虽然只是简单的文本替换，但它全大写的形式非常醒目，可以很容易地识别出名字空间开始和结束的位置。


条件编译（#if/#else/#endif）

利用 “#define” 定义出的各种宏，我们还可以在预处理阶段实现分支处理，通过判断宏的数值来产生不同的源码，改变源文件的形态，这就是 **“条件编译”**。

条件编译有两个要点，一个是条件指令 “#if”，另一个是后面的“判断依据”，也就是定义好的各种宏，而**这个“判断依据”是条件编译里最关键的部分**。

通常编译环境都会有一些预定义宏，比如 CPU 支持的特殊指令集、操作系统 / 编译器 / 程序库的版本、语言特性等，使用它们就可以早于运行阶段，提前在预处理阶段做出各种优化，产生出最适合当前系统的源码。

你必须知道的一个宏是 “__cplusplus”，它标记了 C++ 语言的版本号，使用它能够判断当前是 C 还是 C++，是 C++98 还是 C++11。你可以看下面这个例子。

 复制代码

```
1  #ifdef __cplusplus                                // 定义了这个宏就是在用C++编译
2      extern "C" {                                  // 函数按照C的方式去处理
3  #endif
4      void a_c_function(int a);
5  #ifdef __cplusplus                                // 检查是否是C++编译
6      }                                              // extern "C" 结束
7  #endif
8
9  #if __cplusplus >= 201402                          // 检查C++标准的版本号
10     cout << "c++14 or later" << endl;           // 201402就是C++14
11 #elif __cplusplus >= 201103                       // 检查C++标准的版本号
12     cout << "c++11 or before" << endl;         // 201103是C++11
13 #else // __cplusplus < 201103                    // 199711是C++98
14     # error "c++ is too old"                     // 太低则预处理报错
15 #endif // __cplusplus >= 201402                 // 预处理语句结束
```

除了 “__cplusplus”，C++ 里还有很多其他预定义的宏，像源文件信息的 “**FILE**” “**LINE**” “**DATE**”，以及一些语言特性测试宏，比如 “__cpp_decltype” “__cpp_decltype_auto” “__cpp_lib_make_unique” 等。

不过，与优化更密切相关的底层系统信息在 C++ 语言标准里没有定义，但编译器通常都会提供，比如 GCC 可以使用一条简单的命令查看：

 复制代码

```
1  g++ -E -dM - < /dev/null
```

```

2  #define __GNUC__ 5
3  #define __unix__ 1
4  #define __x86_64__ 1
5  #define __UINT64_MAX__ 0xffffffffffffffffUL
6  ...
7
8

```

基于它们，你就可以更精细地根据具体的语言、编译器、系统特性来改变源码，有，就用新特性；没有，就采用变通实现：

 复制代码

```

1  #if defined(__cpp_decltype_auto)           //检查是否支持decltype(auto)
2      cout << "decltype(auto) enable" << endl;
3  #else
4      cout << "decltype(auto) disable" << endl;
5  #endif //__cpp_decltype_auto
6
7  #if __GNUC__ <= 4
8      cout << "gcc is too old" << endl;
9  #else // __GNUC__ > 4
10     cout << "gcc is good enough" << endl;
11 #endif // __GNUC__ <= 4
12
13 #if defined(__SSE4_2__) && defined(__x86_64)
14     cout << "we can do more optimization" << endl;
15 #endif // defined(__SSE4_2__) && defined(__x86_64)
16

```

除了这些内置宏，你也可以用其他手段自己定义更多的宏来实现条件编译。比如，Nginx 就使用 Shell 脚本检测外部环境，生成一个包含若干宏的源码配置文件，再条件编译包含不同的头文件，实现操作系统定制化：

 复制代码

```


1  #if (NGX_FREEBSD)
2  #   include <ngx_freebsd.h>
3
4  #elif (NGX_LINUX)
5  #   include <ngx_linux.h>
6
7  #elif (NGX_SOLARIS)
8  #   include <ngx_solaris.h>
9
10 #elif (NGX_DARWIN)

```



```
11 # include <ngx_darwin.h>
12 #endif
```

条件编译还有一个特殊的用法，那就是，使用“#if 1” “#if 0”来显式启用或者禁用大段代码，要比“/* ... */”的注释方式安全得多，也清楚得多，这也是我的一个“不传之秘”。

 复制代码

```
1 #if 0           // 0即禁用下面的代码，1则是启用
2     ...         // 任意的代码
3 #endif         // 预处理结束
4
5 #if 1           // 1启用代码，用来强调下面代码的必要性
6     ...         // 任意的代码
7 #endif         // 预处理结束
```

小结

今天我讲了预处理阶段，现在你是否对我们通常写的程序有了新的认识呢？它实际上是混合了预处理编程和 C++ 编程的两种代码。

预处理编程由预处理器执行，使用 #include、#define、#if 等指令来实现文件包含、文本替换、条件编译，把编码阶段产生的源码改变为另外一种形式。适当使用的话，可以简化代码、优化性能，但如果是“炫技”式地过分使用，就会导致导致代码混乱，难以维护。

再简单小结一下今天的内容：

1. 预处理不属于 C++ 语言，过多的预处理语句会扰乱正常的代码，除非必要，应当少用慎用；
2. “#include”可以包含任意文件，所以可以写一些小的代码片段，再引进程序里；
3. 头文件应该加上“Include Guard”，防止重复包含；
4. “#define”用于宏定义，非常灵活，但滥用文本替换可能会降低代码的可读性；
5. “条件编译”其实就是预处理编程里的分支语句，可以改变源码的形态，针对系统生成最合适的代码。

课下作业

最后是课下作业时间，给你留两个思考题：

1. 你认为宏的哪些用法可以用其他方式替代，哪些是不可替代的？
2. 你用过条件编译吗？分析一下它的优点和缺点。

欢迎你在留言区写下你的思考和答案，如果觉得对你有所帮助，也欢迎分享给你的朋友，我们下节课见。

课外小贴士

- 1.C++17引入了一个新的预处理工具“__has_include”，可以检查文件是否存在，注意，不是检查文件是否已经被包含。
- 2.有的编译器支持指令“#pragma once”，也可以实现“Include Guard”，但它是非标准的，不推荐使用。
- 3.C++20新增了“模块”（module）特性，可以实现一次性加载，但“Include Guard”在短期内还是无可替代的。
- 4.使用boost.preprocessor库可以实现复杂的预处理元编程，它提供分支、迭代等基本语言结构，甚至还有数组、链表等容器。

6月-7月课表抢先看

充 ¥500 得 ¥580

赠「¥ 118 月球主题 AR 笔记本」



【点击】图片, 立即查看>>>

© 版权归极客邦科技所有, 未经许可不得传播售卖。页面已增加防盗追踪, 如有侵权极客邦将依法追究其法律责任。

上一篇 02 | 编码阶段能做什么: 秀出好的code style

下一篇 04 | 编译阶段能做什么: 属性和静态断言

精选留言 (25)

写留言



Carlos

2020-05-12

先回答老师的问题 😊

1. 就中间那个算立方体的例子来说, 我觉得这个用法可以替换成一个 inline function.

储存常数可以直接用 const, 为了简化可以用 reference

2. 说实话, 我没用过条件编译 😞, 今天头一次学到, 但是我觉得这肯定会让代码变得更加复杂, 冒出很多意料之外的 bugs. ...

展开 ∨

作者回复:

1.对, 在C++里大部分函数宏都可以用inline函数代替。

2.条件编译多用在跨平台和高级优化方面, 不用当然也可以, 但要追求性能极致就必须考虑。

3.cmake不了解，其实预处理就是文本替换，你要是愿意，用shell、Python也可以处理C++源码来实现。

4.extern "C" {}是处理编译器的链接符号，保持与C兼容，通常是为了导出外部接口使用的。暂时不了解也没关系，以后用到的时候再学也来得及。

1

4



嵇斌

2020-05-12

1. #define PI (3.14) -> constexpr float PI = 3.14 其他貌似不能完全对等。比如内链的代码块，可以用lambda，但是效率如何得看编译器的优化了。比如有些条件编译可以想办法用enable_if来替换实现。到了后面这些替换就不单单是语言方面的实践了，可能涉及软件工程、设计模式。

2. 条件编译自己用还好，自己一般都还清楚自己的套路。条件编译最头疼的就是对着代...

展开 ∨

作者回复: 说的很好。

1.constexpr是编译期的常量，和预处理期的常量效果等价，但生命周期不同。

2.C++对同一个问题有多种解法，虽然自由，但也有选择困难症，挑出合适自己的就比较累。

3.这个也是追求性能付出的代价吧，怕的就是四处散落的#if-#else-#endif，最好把平台相关的代码集中在一起，再用#include一次包含，可以参考Nginx，它做的很好。

3

3



Tedeer

2020-05-12

在读到这段话时：另存为一个 "*.inc" 文件，然后再用 "#include" 替换原来的大批数字。

想起以前开发过程中，曾经在头文件中定义了一个240*160的图片字节数组，现在看来有点蠢，又涨知识了。条件编译在Android系统源码见得比较多，区分不同平台之间代码块的实现。

展开 ∨

作者回复: 预处理就是面向源码编程，调整源码的形态，掌握了这一点就可以让代码更干净整齐。

1

2



sugar

2020-05-21

老师，能否讲讲#include 各种头文件的细节，比如尖角和引号 在include时就是不同的。

作者回复: 这个区别就是包含时的搜索路径不同，网上资料都有，所以就不重复了。好像 "" 是从当前开始搜索，<>是从系统路径开始搜索。

1

1



EncodedStar

2020-05-18

函数式的方法可以用 inline function，内联inline类似于宏，使用inline时，代码在执行前，编译器先将调用的inline函数替换成那个函数的执行代码。

#if endif 这个是复只能制用宏来做百的 其他方式都不行另外，度编译命令行传问递变量值也只能通过宏来做。

之前写过嵌入式程序，条件编译见得比较多，程序在不同系统下都需要运行时就要条件...

展开

作者回复: 说的很好。

1

1



极客时间

2020-05-15

老师 rust很安全，现在大公司很多项目都用rust改写了，cpp还需要学吗，是不是直接学rust呢

作者回复: 可以看一下tobie榜单，C++还在前五，很多框架、系统都是用C++写的，学习它可以更好地理解系统架构。

当然最后你不一定在工作中用到C++，但它作为一个基础，会受用终身的。

1

1



Geek_bc5665

2020-05-14

program once 不是也可以防止重复包含?

展开 ▾

作者回复: 可以, 但不具有普适性, 一般不推荐使用。

1

1



廖熊猫

2020-05-12

我的一些使用经历:

1. 使用`#ifdef __cplusplus`这个在用Emscripten编译wasm的时候会跟`extern "c" {}`这个一起使用, 防止编译后名字被修改掉。
2. 在C语言里直接使用`const`定义的长度在全局定义数组会报错, 但是可以用预处理器来创建。...

展开 ▾

作者回复:

1. 条件编译加`__cplusplus`经常用在与C配合工作方面, 算是个惯用语了。
2. C语言对`const`的支持比较弱, 在C++中好很多。
3. 宏在减少重复代码方面有时候还是挺有用的, 因为模板和泛型的语法检查很严格, 要写好还是要费些力气的, 如果是简单的工作用宏就会轻松一些。

1

1



wuwei

2020-05-12

说实话, 这节课对于cpp小白来讲太抽象了, 很多专业术语是没有接触过的, 应该往后挪一挪的

作者回复: 嗯, 预处理这块是学C++的一个难点, 因为跟C++其实没多大关系, 但确实又很有用。

把它放在前面是因为它与C++程序的生命周期密切相关, 了解它才能更好理解C++, 也能够帮你看懂别人写的C++代码。

如果觉得难, 不好学也没关系, 只要先有个大概了解就好, 后面也会尽量少用宏, 以后在实践中再逐渐应用体会。

**Eason Tai**

2020-05-24

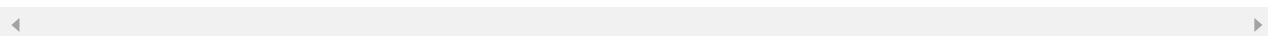
```
#elif __cplusplus >= 201103 // 检查C++标准的版本号  
    cout << "c++11 or before" << endl; // 201103是C++11
```

为什么不是 c++11 or later 呢

展开 ∨

作者回复: 这个好像是笔误, 写错了, 汗啊。

先领会条件编译的用法吧。

**sugar**

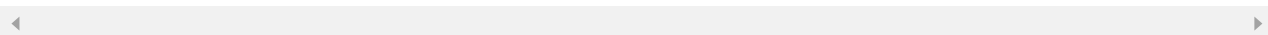
2020-05-21

另一个有关include头文件的问题。比如说我看很多例子都是 在单独一个h文件和一个cpp文件中定义了一个类, h里写了类的声明, 类的具体实现在cpp里面。其他cpp文件里想引入这个类的时候, 就直接include了h文件, 并没include cpp文件, 按照咱们这一讲 只有声明被预处理时放到实际执行的cpp代码里了, 类的实现cpp是怎么引入进来的呢?

展开 ∨

作者回复: h里一般只是声明, cpp里是实现, 在编译阶段会有个链接动作, 把这些都链接起来。

具体的细节比较复杂, 我也没细研究过, 可以网上找点资料看看, 不过我觉得不了解也没关系。

**有学识的兔子**

2020-05-16

学习c++有些时间了, 对c++几个版本是基于时间命名才了解到, 有必要了解这些版本的大致区别。

1. 宏不可替代的地方: 基于它们早于运行阶段, 例如一些操作是需要运行前完成的; 只读变量可以用const, 类型别名typedef。

2. 特别不喜欢有人在类的成员函数里用条件编译来作为代码的分支路径使用, 给人感觉...

展开 ∨

作者回复: 条件编译通常是用来处理系统、平台差异的, 如果要像你说的那么使用确实是有点过分。



Soda

2020-05-15

第一个问题, 能想到的就是定义常量用const代替, 因为编译器会替我们检查类型。

第二个问题, 用过条件编译, 很适合在开发嵌入式程序的时候, 做一些跨平台的处理 😊

作者回复: 说的很好。



锦鲤

2020-05-14

想找些课程练手写C++代码, 最好是开源的, 有答案的那种, 罗老师有推荐吗?

Linux下C++编程方面的书, 能否先推荐几本? 中级水平的那种

作者回复:

1.C++开源项目很多都比较大, 练手级别的少, 我没有太关注, 可能要你自己去GitHub上找一下了。

可以先参考课程GitHub的readme, 里面有很多链接。

2.C++经典书比较少, Linux+C++就更少了, 我没有看到特别好的。

其实你应该分开来看, 学C++, 再学Linux, 而不是Linux+C++。

后面我会有个经典书推荐, 学好C++, 再来一本《UNIX环境高级编程》就差不多了。



忆水寒

2020-05-14

前一段时间刷了点leetcode题目, 用C++写的。

https://mp.weixin.qq.com/s/LYWEFFDVjUX6N48daq_QBQ

展开 ∨

作者回复: LeetCode太高端了, 我那时候都没见过, 感觉刷LeetCode的人都很厉害。



jxon-H

2020-05-14

这节课真是干货满满，我从来都不知道预处理阶段包含这么多知识，我还活在那个以为 # 就是只是和 include 搭配的世界。

预处理阶段内容是那么的充实丰满.....

我满篇文稿的划线标笔记，但是回个头认真思考，这些丰富的经验总结，岂是几根线就可以消化的吗？ ...

展开 ▾

作者回复: 预处理是C++的重要组成部分，很强大也很有意思，但也很难用好。

C++迷人之处就在这里，很多特性可以深挖，内涵丰富，但小心不要过度沉迷，一定要记得代码是写给别人看的。



1



yelin

2020-05-13

老师真的经验丰富啊，赞

展开 ▾

作者回复: 这些都是写代码看代码多了总结出来的，希望能够帮助你少走些弯路。



九三

2020-05-13

老师，用的这些像 `__cplusplus` `__GNUC` `__VERSION` 等，这些关键字是系统提供的还是代码提供的？

作者回复: 有的是标准定义的，有的是编译器定义的。



1



九三

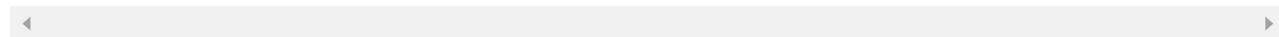
2020-05-12

`g++ test03.cpp -E -o a.cxx`

老师这个输出预处理的文件， a.cxx ,是特定的吗，我使用a.txt 也可以吗

展开 ▾

作者回复: 后缀名随意起，当然a.txt也可以。



hb

2020-05-12

Effective Objective-c 2.0 第4条： 多用类型常量， 少用#define预处理指令，请问下老师这个是否有什么讲究，还是说code style?

展开 ▾

作者回复: 它说的没错，应该多用const，少用宏，但不是说完全禁止使用。

适当使用宏也可以提高代码的可读性和运行效率，多在写代码的过程中实践，就能够找到最合适你自己的“度”。

